AN EMPIRICAL ANALYSIS OF IOT MALWARE INFECTION TECHNIQUES

A Dissertation
Presented to
The Academic Faculty

by

Nicholas Joaquin

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
May 2020

AN EMPIRICAL ANALYSIS OF IOT MALWARE INFECTION TECHNIQUES

Approved by:

Dr. Manos Antonakakis, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Angelos Keromytis
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Elliot Moore II
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: April 29, 2020

ACKNOWLEDGEMENTS

I wish to thank my mentor, Omar Alrawi, as well as the head of the Astrolavos Lab, Dr. Manos Antonakis. I would also like to thank Dr. Amanda Madden for her guidance and help with the research. Finally, I would like to thank my parents, Erick Joaquin and Anita Joaquin, for their continuous support in every aspect of my life.

TABLE OF CONTENTS

LIST OF TABLES AND FIGURES

ABSTRACT

The rise of insecure Internet of Things (IoT) on the Internet is problematic because they are easily compromised. IoT vendors are trying to push products to market as quickly as possible resulting in a significant amount of security issues. This work explores the attacks vectors used by malware to gain privilege control of IoT devices. We achieve this by performing two experiments – a static binary analysis that checks for specific patterns and identifies a binary to a publicly disclosed vulnerability, and a dynamic binary analysis focusing on linking program behavior to malicious actions. We further extend upon this by analyzing ELF section metadata of "tagged" binaries to determine if we can link specific ELF section sizes and entropies to malicious binaries. Through our work, we see that a large portion of vulnerabilities occurs due to improperly validated inputs, followed by weak credentials and improperly secured files. Moreover, we have also found that we are unable to link ELF section metadata to malicious binaries, as a result of anti-analysis efforts by malware authors. Our intention with this work is to understand how malware attacks IoT devices, thereby highlighting the specific security areas that must be prioritized in IoT device development.

CHAPTER 1. INTRODUCTION

Between 2016 and 2018, household Internet of Things (IoT device, e.g. smart fridges, Google Home) adoption rate rose 48%, from 4.7 billion devices to 7.0 billion devices connected to the Internet globally [1,2]. The residential IoT market has recently established itself as a multi-billion-dollar industry [3], evolving into an extremely lucrative field for both new and established companies. With an influx of new IoT products being created to keep up with the market demand, IoT software security has largely been overlooked. This has left consumers compromised to attacks such as botnet opt-in [4], man-in-the-middle attacks [5], and illicit surveillance [6].

Ample studies exist on the topic of IoT malware and cover many methodologies from how to secure deployed IoT devices on a local network [7] to how to analyze IoT devices for security research purposes [8]. These studies focus on securing and isolating a network with IoT devices from external attackers. However, with new and evolving threats arising as a result of global device adoption, it can be difficult to adapt to attacks without knowing attack vectors. By knowing current attack vectors, vendors and developers will be able to prioritize what to protect in developing IoT devices.

In this study, we present our methodology in analyzing IoT Malware to determine current attack vectors, utilizing static and dynamic binary analysis techniques to discover and characterize possible malicious behavior of a binary. We make use of malware tagging using *Yara* signatures to scan a binary for specific patterns linked to publicly disclosed vulnerabilities, as well as make use of *strace* to capture and summarize a binary's behavior. Moreover, we make use of Python modules *Lief* to analyze ELF section metadata to

determine if it is possible to link specific metadata values to malicious binaries. Through our work, we were able to identify the attack vectors of 8410 out of 42327 binaries relating to 56 publicly disclosed vulnerabilities. Moreover, we were also able to determine that correlation between ELF section metadata and malicious binaries is not possible.

CHAPTER 2. LITERATURE REVIEW

Current research around malware in IoT devices primarily revolves around two fields – security framework analysis and security standard proposals. For security framework analysis, research primarily focuses on methods to secure and isolate IoT devices at a network level to prevent malware infection and/or propagation within large-scale systems. Meanwhile, security standard proposals focus on advocating for a software and hardware policy that will be globally implemented and "impenetrable."

Riahi, A., et. al [7] proposed a method in theoretically securing an IoT device within a network or ecosystem with user-centrical security analysis, focusing on a user's interaction with an IoT device. This paper, more importantly, sets out a single framework to analyze malware with its "tension-recommendation" model. In this model, a researcher determines possible faults of the security (the tension) and any adjustments that can be made to prevent any security problems (the recommendation). In a more technically-driven analysis, Mare, S., et. al [9] evaluates seven popular IoT devices and assesses each device based on security within access control, privacy, automation, and interoperability. The Mare paper also makes use of this "tension-recommendation" model but extends this model to cater to specific devices and technical methods. However, researchers cannot find common ground on specific IoT security problems. For example, the Mare paper recommends that to maintain privacy within an IoT device a device must reduce leaks from side-channels – functions that are not core to the IoT device. However, Sivaraman, V., et al. [10] instead recommends using an SMP protocol to prevent any data leaks of any private data. More recently, researchers have released Proof of Concepts to supplement their

research of framework analysis, for example in the papers of Alrawi, O., et al. [8] and Fernandes, E., et al. [11]. With no common ground, research can only go so far as to be "soft recommendations" within the IoT field. As the malware landscape continues to evolve, papers and frameworks such as these can quickly become obsolete.

More recently, to achieve consistency between IoT platforms, researchers have begun to propose standards for security frameworks, as opposed to adding on to pre-existing proprietary IoT platforms. For example, Lake, D., et al.'s study [12] proposes an architectural security framework standard for IoT devices within the eHealth field. The paper recommends building a framework based on a singular "building block stack," so that when a security issue arises within an eHealth device, the problem area can easily be pinpointed and fixed. This prevents any issue with vendors having to work through proprietary frameworks to mitigate a problem. While this is very much a valid solution, the main issue that hampers its implementation is the difficulty of collectively agreeing upon a standard. Two papers that demonstrate this are Goutam, S., W. Enck, and B. Reaves, [13] and OConnor, T., et al. [14]. The Goutam-Enck-Reaves paper proposes "Hestia" – a security standard primarily focused on simple isolation policies where controllers (such as Smart Hubs) are only connected to the global network and non-controllers (such as Hue Light bulbs) are only connected to controllers. Meanwhile, the OConnor paper proposes "HomeSnitch" – a security standard that separates an IoT device's core functionality from its semantic behavior, hardening security on specific behavioral traits rather than the products itself. However, almost ironically, these two papers presenting different standards for IoT security were presented at the same conference. This demonstrates that in creating a standard for security in a billion-dollar market, no one can agree on a specific standard.

For our work, we decided to step away from higher-level security analysis and instead look at vectors currently used to attack and infect IoT devices. Our reasoning behind identifying attack vectors is to look at network IoT security at a device level rather than at a holistic network level. By looking at attack vectors within the IoT device, we can complement current security research in IoT and leverage traditional security defense ecosystems to assist vendors in proactively protecting devices from attackers.

CHAPTER 3. METHODOLOGY

In the analysis of embedded Linux binaries, there are two significant analysis methods we considered in determining how a malicious Linux program infects and compromises IoT devices. These techniques are *Static Binary Analysis* and *Dynamic Binary Analysis.* We extend upon these by analyzing *ELF Section Metadata* to determine if there is any correlation between malicious binaries and ELF section metadata.

3.1 Static Binary Analysis

The primary purpose of conducting static binary analysis is to determine what infection vectors Linux binary uses to infect an IoT device. Static binary analysis made use of *Yara signatures* to identify if a binary uses a known vulnerability for infection. *Yara* is a program that takes in signatures (metadata files that contain a tag that is paired to a matchable hex or string pattern) and analyzes a set of binary samples matching a binary to a specific tag.

3.1.1 Infection Detection

There are two main ways that we produced signatures to tag binaries in Yara: manual generation with addition verification vetting (sourced from NVD and Routersploit) and automatic generation using Alienvault OSSim rules.

3.1.1.1 Manual Generation – Sourcing from NVD

To create our *Yara Signatures*, we first compiled a list of vendors that produced IoT devices. This vendor list comprised both of companies that specialize in (such as Lutron,) as well as larger technology companies which produce IoT devices as part of their product portfolio (e.g. Google, Amazon, etc.)

Once this list of vendors was compiled, we scraped all CVEs whose vulnerable product was produced by a company in our vendor list. To do this, we downloaded the JSON-formatted data feeds from 2002 to 2019 from the National Vulnerability Database of CVEs [15]. We then went through every CVE entry and looked at the Common Platform Enumeration v2.3 Uri (cpi23Uri) string for each product vulnerable to the specific CVE. If the fourth field of the cpi23Uri string (corresponding to a vendor) contained our target vendor, we placed the CVE in our database.

After compiling the list of all the CVEs that correspond to our vendor list, we filtered out all CVEs that did not apply to IoT or networking based devices. Since our list of vendors consisted of larger companies that produce non-IoT and non-networking products, we had to remove all CVEs that were associated with irrelevant software and hardware. To do this, we compiled a list of products grouped by vendor by iterating through the aforementioned CVE data feeds and associated cpi23Uri strings. We then scraped the fifth field of each cpi23Uri string corresponding to the vulnerable product, grouping the product with the previously retrieved vendor. Once this list of products was compiled, we manually looked through the list of products to determine which products were relevant for our research, removing non-IoT and non-networking products from our list. With this

final list of relevant products, we iterate through the data feed again to retrieve a list of CVEs that pertain to IoT products.

To further filter out what CVEs we can write Yara signatures for, we iterated through the data feeds one last time to compile a list of CVEs with its associated Proof of Concept (PoC) exploit code. To retrieve the PoC code, for each CVE in the data feed we check if there is an exploit-db.com link within the "references" field. If there is, we access the link and scraped the PoC exploit code from the website source code. If the CVE does not have PoC exploit code or an exploit-db.com link within its references, we remove it from our list of CVEs.

To write our Yara signatures, we looked at each PoC exploit code to determine if a unique and discernable pattern existed that could link a binary to a specific CVE. If a CVE did not have a unique and discernable pattern, such as that in Denial of Service and Buffer Overflow vulnerabilities, we removed it from our list of CVEs. We then wrote our Yara signature in the Yara file format

From our first-generation process, we scraped 365 PoC exploit codes, which we filtered down to create 69 YARA signatures.

3.1.1.2 Manual Generation – Sourcing from Routersploit

To generate Yara signatures from Routersploit modules, we first pulled a list of exploits from the Routersploit GitHub repo [16]. When looking at each module, we see that each PoC exploit code is provided to us as python code. For each PoC python script, we looked through the code to determine if there was a unique string or a group of unique

strings that could be linked to a specific Routersploit vulnerability (determining string uniqueness is outlined in 3.1.1.1)

If it is possible to write a Yara signature, we manually write the Yara signature in the Yara file format using the unique group of strings. Otherwise, we do not include the Routersploit module in our database of Yara signatures. Through this process, we wrote 113 signatures.

3.1.1.3 Automatic Generation – Sourcing from Alienvault's OSSim Emerging Rules

Alienvault's OSSim is a security information and event management (SIEM) system that, when given a set of rules, can detect, filter and block network traffic in real-time. In this section, we are automatically generating Yara signatures given a set of OSSim rules pertaining to network devices.

To automatically convert an OSSim rule to a Yara Signature, we first looked at the structure of an OSSim rule. Through analysis, we see that a rule is in the following structure:

alert [protocol] [source IP]->[dest IP] (msg:"[tag]"; content:"[signature]";
content:"[signature]; ...)

For our research, the most interesting fields and contents for us were the "msg" field and "content" fields of the rule, corresponding to the tag and the signature of the rule. In generating our Yara signature, we made use of these fields to write our signature in the Yara file format.

One quirk of an OSSim rule that is not present in YARA signatures is that the "content" of the rule mixes hex values and ASCII characters, switching between the two with the pipe (|) character. Such is the case in the rule "ET EXPLOIT Pwdump3e pwservice.exe Access port 445," which has the content of "p|00|w|00|s|00|e|00|r|00|v|00|i|00|c|00|e|00|.|00|e|00|x|00|e." As observed, this is just the string "pwservice.exe" in wide-character format. Since this cannot be done in a Yara signature, we had to convert all ASCII characters of these "mixed strings" into hex values before writing the Yara signature.

For our research, we performed the automatic generation of Yara signature on a database of router rules called "emerging-exploit," [17] gathering 254 automatically-generated Yara-signatures

3.1.2 Signature Evaluation

Before running the Yara signatures to collect data, we verified correctness by running our signatures on a set of 42,327 malicious binaries. These malicious binaries were sourced from Virus Total.

To verify these Yara signatures, we first wrote a "master" Yara signature that made use of Yara's "include" syntax to combine all the Yara signatures into a single file [18]. Using this master rule, we then made use of Yara to scan and tag every binary, piping output into a text file. Scanning 42,327 binaries with 436 Yara signatures took approximately 45 seconds to complete.

Once we were able to tag every binary, we counted how many times each signature was tagged. We sorted these signature occurrences to gather a general distribution to determine what signatures may be *false positives* and what signatures were *true positives*. Looking at our signature occurrence counts, we flagged any signatures that were tagged more than 10000 times and any signature that was tagged less than 100 times. The reason why these thresholds were chosen was that if the signature's detection pattern is too simplistic, it would tag an unusually large amount or an unusually small amount of times. Out of 436 signatures, we flagged 232 signatures for manual verification

After the flagging process, we looked at all the flagged signatures to determine if the signatures were valid enough to be used for static analysis. If we found that a signature was too simple (ie, a two-byte pattern – one that was automatically generated) we would remove the Yara signature from our database. Otherwise, if the signature was specific but tagging unusually high or unusually low, we looked at the binary using NSA's *Ghidra* to determine if the detection patterns are being correctly utilized to detect the proper vulnerability. We made use of Ghidra's powerful disassembler, tracing tool, and hex search to trace the string pattern to the pattern's usage in code. If we found that the provided detection patterns are being used to detect and exploit the vulnerability, we keep the Yara signature within our database. Otherwise, we remove the signature from our database.

Once we completed this manual verification process, we were confident in moving forward in using the signatures to detect vulnerabilities. After this process, from a total of 436 signatures, we moved forward with 209 signatures to detect vulnerabilities exploited within binaries.

To run Yara, we re-built another master signature comprising of "imports" of our 209 signatures that were validated to detect vulnerabilities. Using this master signature, we proceeded to run Yara against our set of malicious binaries to tag binaries with specific Yara signatures. Again, we count how many times a signature was tagged.

Once we completed the tagging and counting of Yara signatures, we grouped the tagged signatures into four distinct infection vector groups: remote code execution, privilege escalation, information leaks, and miscellaneous. After grouping, we arrive at the table of results present in section 4.2.

3.2 Dynamic Binary Analysis

The primary purpose of conducting dynamic binary analysis is to determine the behavior of a Linux binary and verify the infection technique. A dynamic binary analysis provides us with observed scanning and exploitation and can be utilized to validate the attack vectors and methods based on commonly detected behavioral patterns. Currently, dynamic binary analysis efforts are incomplete but will be utilized as a black box to verify the infection technique.

3.2.1 System Call Tracing

To conduct a dynamic binary analysis on a binary, we run *strace* on the malicious binary to capture a program's behavior. To run *strace* with a target binary, we first determine what architecture the binary using Linux's *file*. Once we know the architecture of the file, we use *qemu* to emulate processor architecture and run strace with the binary

file. We capture the output of *strace* and use this system call trace for further summarization and analysis.

On occasion, binaries may run in an infinite loop, whether to look for targets to propagate to, perform attacks once given a "kill switch." To prevent an infinitely large output file size, we limit *strace* captures to 10 minutes. If a program were to run for 10 minutes, we kill the program.

3.2.2 Trace Classification

Before summarizing system call traces, we first group all possible system calls into six distinct categories: File, Network, Process, Memory, Environment, and Other. To do this, we manually go through all system calls present in Linux's man pages [19] and read each description, allocating each system call into a category that fits its function.

Once we group every system call into the 6 categories, we proceed to clean up and summarize system call traces. To clean up a system call trace, we split each system call within the trace into four distinct sections – the function name, list of arguments, and return value, and time after the program was run (in milliseconds) outputting the data to a JSON format. The clean system call traces were then summarized by counting the system call category occurrences within the program.

3.2.3 OS Object Behavior Modeling

We further extend our system call summarizing by building an OS object dependency graph. First, we manually looked through all system calls again on Linux man pages to record which system calls either created a file descriptor or uses a file descriptor

within the arguments. In this process, we record what the file descriptor created is, as well as the function's file descriptor argument number.

We then iterate through all the clean system calls traces to link an OS object/file descriptor to the function that created it, as well as the functions that use it. Given these sets of functions linked to an OS object, we group and count categories of system calls per OS object to create a category summary per file descriptor.

3.3 Metadata Analysis

We extend upon our positive binary dataset by plotting ELF section metadata to determine whether it is possible to identify a malicious binary based on a binary's section metadata. In this section, we are recording the count, size, and entropy of ELF sections and graphing this data as a cumulative distribution function. We perform this analysis to determine if there is any correlation between ELF section metadata and malicious binaries.

Once we complete malicious binary identification (as per section 3.1), we compile two lists: a list of binaries that were tagged as "positive" by *Yara* (the "hit" group) and a list of all binaries in the malware corpus (the "all" group.) Per binary and group, we use the python tool *Lief* [20] to collate the number of sections in a binary and the size and entropy of each section in the binary. At the end of this collation, we end up with two data sets per group: the number of sections in each binary, and the size and entropy of each ELF section per binary. We proceed to group the latter data set by the ELF section and culminate a list of all sizes and entropies matching the requested ELF section.

From these data sets, we proceed to filter out ELF sections that will not provide conclusive data. We do this by looking at the ELF section data set of the “hit” group and removing any ELF section and respective metadata that has less than 10 appearances in the hit group. By doing this, we filtered our ELF section analysis space from 107 sections to 64 sections.

Upon filtering our sections, we plot cumulative distribution functions (CFs) of all of our data sets. With our data sets, we produced three CDF plots per group: a CDF plot of section counts, a CDF plot of section sizes per ELF section, and a CDF plot of section entropies per ELF section. In the end, we graphed 52 sections for a total of 208 CDF graphs. Including the two CDF graphs of section counts, we compiled 210 CDF graphs of ELF section metadata.

3.4 Technical Challenges

Most of the challenges that occurred in the analysis process surrounded creating high-quality YARA signatures for static binary analysis. As it is difficult to automatically derive usable signatures, we had to manually generate our signatures to ensure correctness.

Our first challenge was in filtering CVEs before manually writing a YARA signature. Since most vendors are not IoT specific, once we collated our list of CVEs linked to each vendor, we had to look at what device each CVE is linked to and remove CVEs which are linked to non-IoT related devices.

After this first filtering process, we then looked at the exploit source code of the CVE to determine if we can write a YARA signature based on unique patterns present in the

source code. If the CVE did not have a provided exploit source code, or there were no unique and specific patterns that could be used to tag a binary to a specific CVE, we removed that CVE from our list.

Our second challenge arose once we built our database of YARA signatures. As we automated the writing of 58% of YARA signatures, it was necessary to test for false positives to get completely correct data. To do this, we ran all YARA signatures against our sample set of 42,327 binaries, counting how many times each signature was found. Once we collated a list of signatures and their "found" counts, we look at which signatures had an unusually high or unusually low "found" count, tagging such signatures as "suspicious."

Amongst all these "suspicious" signatures, we first looked at the criteria patterns to determine if they were specific enough to be used to tag a binary. If the criteria patterns were too simple, we would remove that signature from our database. However, if the criteria patterns looked specific, we analyzed binaries that were tagged with that signature using the Ghidra disassembler to determine whether or not the criteria patterns were valid indicators of the vulnerability. If the binary did use the criteria patterns to exploit the vulnerability, the signature was kept in the database. Otherwise, the signature would be removed.

These challenges tested our abilities to maintain due diligence to retrieve correct results in determining IoT attack vectors. For malware analysis using YARA, we started with a database of 436 Yara signatures**,** of which 58% of signatures were automatically generated and 42% of signatures were manually written. After this filtering and refining process, the

final count of Yara Signatures was 209, of which 85% of signatures were automatically generated and 15% of signatures were manually written.

CHAPTER 4. RESULTS

The results of section 4.1 are retrieved from the plotting of ELF section metadata as described in section 3.3. Our malware samples were gathered from VirusTotal and correspond to binaries found in IoT devices that triggered positive AV detections. Using the malware corpus and tagging results from section 3.1, we are analyzing and plotting size and entropy metadata of binaries, "tagged" or otherwise.

The results of section 4.2 are gathered from the static binary analysis methodology described in section 3.1. In the static analysis, we are using Yara and our database of 209 valid signatures to tag a binary to a specific publicly disclosed vulnerability.

4.1 ELF Header Analysis

Figure 1 shows the CDFs of the number of sections per binary file in the "hit" group (plotted in red) and the "all" group (plotted in green).

Figure 2 is side-by-side comparisons of size and entropy CDFs for random ELF sections. The green plots on each graph represent the CDFs of "all" binaries, while the red plots on each graph represent the CDFs of "all" binaries.

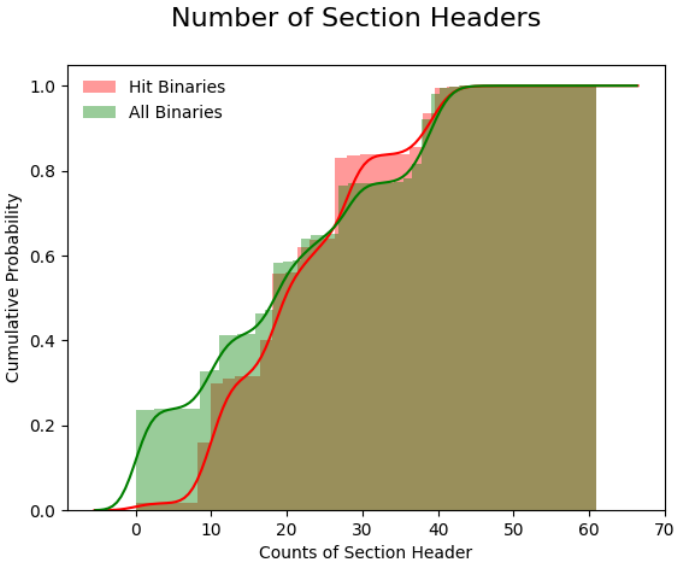

Figure 1: CDF Comparisons of the Number of ELF Sections

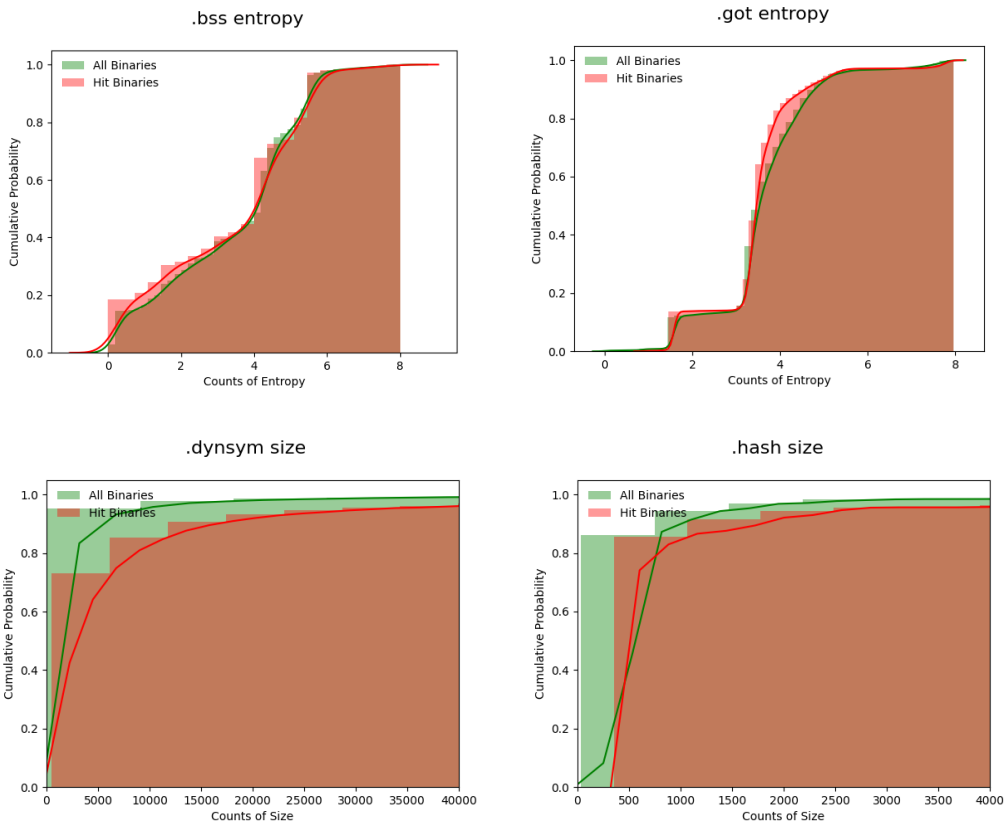

Figure 2: CDF Comparisons of the ELF Section Metadata

4.2 Infection Analysis

Table 1 presents the top 10 Yara signatures by tagged counts. When we ran all our signatures against our corpus of 42,327 binaries, we found that 55 of our 209 signatures (27%) garnered a positive result (a “hit”) and that 8410 binaries (20%) were able to have their infection vectors identified.

Table 2 extends upon Table 1 and groups the signature hit counts into their respective attack vector groups. As we can see, the most common attack vector found was “Remote Code Execution,” comprising of 87% of all tags, followed by “Privilege Escalations” with 12% of all tags, and “Information Leaks” with 1% of all tags.

Vulnerability	Vendor	CVE	Attack Vector Group	# of Tags
HuaweiHomeDeviceUpgrade	Huawei	CVE-2017-17215	Remote Code Execution	4573
realtek_sdk_miniigd_upnp	Realtek	CVE-2014-8631	Remote Code Execution	2246
f6xx_default_root	ZTE		Privilege Escalation	1549
gpon_bypass_injection	Dasan	CVE-2018-10562	Remote Code Execution	1220
linksys_eseries_rce	Linksys		Remote Code Execution	1146
dlink_dsl_2750B_ci	D-Link		Remote Code Execution	996
CVE_2019_9082	ThinkPHP	CVE-20190-9082	Remote Code Execution	600
p660hn_t_v1_rce	Zyxel	CVE-2017-18368	Remote Code Execution	500
hnap_rce	D-Link		Remote Code Execution	471
d1000_rce	Zyxel	CVE-2016-10372	Remote Code Execution	335

Table 1: Top 10 Yara Signatures by Number of Times Tagged

Attack Vector Group	# of Yara Signatures	# of Times Tagged
Remote Code Execution	42	13720
Privilege Escalation	7	1855
Information Leak	6	191

Table 2: Attack Vector Groups by Number of Times Tagged

CHAPTER 5. DISCUSSION

5.1 Findings and Recommendations

Section 4.1 attempts to correlate ELF section metadata to malicious binaries. Figures 1 and 2 consist of samples of CDFs of ELF section metadata, displaying two plots per CDF corresponding to “hit” binaries and “all” binaries (per section 3.3). However, on closer examination, there is no distinguishable difference between the ELF section metadata CDFs of a “hit” binary and an “all” binary for all ELF sections. Thus, it can be demonstrated that we cannot correlate section metadata to malicious binaries.

In observing the results of section 4.2, we can see that the most common attack vector was that of “Remote Code Execution,” or RCE, comprising 76% of hit YARA signatures (42/55) and tagging 13720 times (87% of all tags). The RCE attack vector involves exploiting improperly validated inputs (such as arguments in a GET or POST request) and crafting input to send to an IoT device in order to run a shell command. This can be very dangerous to an IoT device as a single shell command can remotely download and run dangerous code.

The second most common attack vector present was that of “Privilege Escalations,” comprising 13% of hit YARA signatures (7/55) and tagging 1855 times (12% of all tags). In the Privilege Escalation attack vector, an attacker attempts to either brute-force the username and password of the IoT device from a list of default accounts or attack an admit through targeted cross-site request forgery to login. Once an attacker logs in, they can run shell commands to further propagate malware or deploy payloads.

The least common attack vector was that of "Information Leaks," comprising 11% of hit YARA signatures (6/55) and tagging 191 times (1% of all tags.) In this attack vector, an attacker attempts to access specific web pages and configuration files as an unauthorized user to scrape information on admin login credentials and machine settings. Once admin credentials are found, an attacker logs in to the device and run shell commands as described above. This attack vector arose due to improperly set permissions for files that contain private information.

5.2 Future Work

Future work in this research area will center on dynamic binary analysis and labeling behavior given summarized traces of a binary's system calls. Our current work in dynamic binary analysis involves summarizing system calls at a high level of abstraction and building a chain of analysis to link specific system calls to different OS objects.

Future work in dynamic binary analysis will involve categorizing and labeling the behavior based on the system call summaries to verify the vulnerability by the malware to propagate. An extension to this work will be to employ machine learning to tag binary behavior automatically given system call summaries, instead of through manual analysis. The purpose of dynamic binary analysis and conducting behavior labeling is to observe network scanning and exploitation. By pairing dynamic binary analysis with static binary analysis, we can garner a holistic report on the activity of a binary and the currently applied methods used by malware to infect IoT devices.

Other areas of future work that can be taken extend upon the work performed on ELF section metadata analysis. An interesting observation that we encountered during this

process was the existence of a significant amount of “debug” ELF sections, comprising 18 out of 52 sections analyzed (~35%). These sections appear in binaries as a result of improper compilation and are usually artifacts of toolkits. Future work in this area can look into the existence of specific sections with ELF binaries and determining a formative relationship between specific ELF sections and malicious binaries.

CHAPTER 6. CONCLUSION

In this work, we determine the attack vectors currently used by IoT malware to propagate between devices and spread throughout a network. We perform three main analyzes on a malware corpus of 42 thousand binaries to achieve this task: a static binary analysis comprised of Yara based pattern matching, a dynamic binary analysis based on summarizing system call traces, and an ELF section metadata analysis. Based on our work, we can determine that the most common attack vector currently used is the “remote code execution” vulnerability. This is followed by the “privilege escalation” attack vector, and finally by the “information leak” vector. Lastly, we proceed to graph ELF section metadata to find a pattern between section metadata and malicious binaries. However, this section of work does not produce any formative conclusions. Future studies could further extend upon dynamic binary analysis and identify malicious binaries through suspicious behavior patterns.

REFERENCES

1. Lueth, K.L. *State of the IoT 2018: Number of IoT devices now at 7B – Market accelerating*. 2018 [cited 2019 July]; Available from: https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/.

2. Kaspersky. *New IoT-malware grew three-fold in H1 2018*. 2018; Available from: https://www.kaspersky.com/about/press-releases/2018_new-iot-malware-grew-three-fold-in-h1-2018.

3. McKinsey. *Unlocking the potential of the Internet of Things*. 2015 [cited 2019 June]; Available from: https://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/the-internet-of-things-the-value-of-digitizing-the-physical-world.

4. Antonakakis, M., et al., *Understanding the mirai botnet*, in *Proceedings of the 26th USENIX Conference on Security Symposium*. 2017, USENIX Association: Vancouver, BC, Canada. p. 1093-1110.

5. Cimpanu, C. *Over 65,000 Home Routers Are Proxying Bad Traffic for Botnets, APTs*. 2018 [cited 2019 June]; Available from: https://www.bleepingcomputer.com/news/security/over-65-000-home-routers-are-proxying-bad-traffic-for-botnets-apts/.

6. Williams-Grut, O. *Hackers once stole a casino's high-roller database through a thermometer in the lobby fish tank*. 2018 [cited 2019 June]; Available from: https://www.businessinsider.com/hackers-stole-a-casinos-database-through-a-thermometer-in-the-lobby-fish-tank-2018-4.

7. Riahi, A., et al. *A Systemic Approach for IoT Security*. in *2013 IEEE International Conference on Distributed Computing in Sensor Systems*. 2013.

8. Alrawi, O., et al., *SoK: Security Evaluation of Home-Based IoT Deployments.* 2019.

9. Mare, S., et al., *Consumer Smart Homes: Where We Are and Where We Need to Go*, in *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*. 2019, ACM: Santa Cruz, CA, USA. p. 117-122.

10. Sivaraman, V., et al. *Network-level security and privacy control for smart-home IoT devices*. in *2015 IEEE 11th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. 2015.

11. Fernandes, E., et al., *Security Implications of Permission Models in Smart-Home Application Frameworks.* IEEE Security & Privacy, 2017. **15**(2): p. 24-30.

12. Lake, D., et al., *Internet of Things: Architectural Framework for eHealth Security*. Vol. 1. 2014.

13. Goutam, S., W. Enck, and B. Reaves, *Hestia: simple least privilege network policies for smart homes*, in *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*. 2019, ACM: Miami, Florida. p. 215-220.

14. OConnor, T., et al., *HomeSnitch: behavior transparency and control for smart home IoT devices*, in *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*. 2019, ACM: Miami, Florida. p. 128-138.

15. National Institute of Standards and Technology (NIST). *NVD - Data Feeds,* 2020 [cited 2020 March]; Available from: https://nvd.nist.gov/vuln/data-feeds

16. mcw0. *routersploit/exploits,* 2020 [cited 2020 March]; Available from: https://github.com/mcw0/routersploit/tree/master/routersploit/modules/exploits

17. jpalanco. *alienvault-ossim/emerging-exploit,* 2020 [cited 2020 March]; Available from: https://github.com/jpalanco/alienvault-ossim/blob/master/suricata-rules-default-open/rules/1.3.1/emerging.rules/emerging-exploit.rules

18. readthedocs. *Including Files,* 2020 [cited 2020 March]; Available from: https://Yara.readthedocs.io/en/v3.4.0/writingrules.html#including-files

19. die. *Section 2: system calls - Linux man pages,* 2020 [cited 2020 March]; Available from: https://linux.die.net/man/2/

20. QuarksLab. *Lief,* 2020 [cited 2020 April]; Available from: https://lief.quarkslab.com/